Machine Learning for NLP
Lecture 2: Linear classifiers

Richard Johansson

UNIVERSITY OF GOTHENBURG

September 2, 2016

math in machine learning

- machine learning is a "mathy" subject...
- the most important branches of mathematics used in ML:
  - probability and statistical theory
  - linear algebra
  - optimization
- in this lecture, we'll see some basic **linear algebra** and see how that relates to classifiers
- we will also take a look at how linear algebra is implemented in Python and use it to code a learning algorithm

overview

the perceptron revisited

basic linear algebra and its implementation in Python

converting features to numerical vectors

linear classifiers

the perceptron classifier

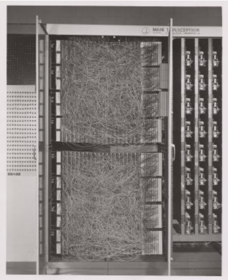
- the perceptron learning algorithm creates a **weight table**
- each weight in the table corresponds to a feature
  - e.g. `"fine"` probably has a high positive weight in sentiment analysis
  - `"boring"` a negative weight
  - `"and"` near zero
- classification is carried out by summing the weights for each feature
  - one class is associated with positive scores, another with negative scores
  - so we can handle two-class problems: **binary** classification

the perceptron learning algorithm

- start with an empty weight table
- classify according to the current weight table
- each time we misclassify, change the weight table a bit
  - if a positive instance was misclassified, add 1 to the weight of each feature in the document
  - and conversely ...

a historical note

- the perceptron was invented in 1957 by Frank Rosenblatt
    - here's an image (from Wikipedia) of the first implementation
- initially, a lot of hype!
- the realization of its limitations led to a backlash against machine learning in general
    - the nail in the coffin was the publication in 1969 of the book *Perceptrons* by Minsky and Papert
- new hype in the 1980s, and now...



overview

the perceptron revisited

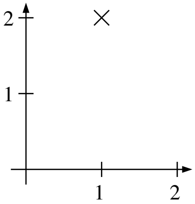
basic linear algebra and its implementation in Python

converting features to numerical vectors

linear classifiers

vectors

- a tuple consisting of n numbers is called a **vector**
- the set of all possible tuples of length n is called an n-dimensional **vector space**
- for instance: $(1, 2)$ is a 2-dimensional vector
- they can be interpreted geometrically, either as a point in a coordinate system



- ...or as a direction (e.g. of motion or force)

basic linear algebra

the basic operations on vectors:

- **scaling**: $\alpha \cdot \boldsymbol{v} = \alpha \cdot (v_1, \ldots, v_n) = (\alpha \cdot v_1, \ldots, \alpha \cdot v_n)$
- **addition** and **subtraction**:
  $\boldsymbol{v} + \boldsymbol{w} = (v_1, \ldots, v_n) + (w_1, \ldots, w_n) = (v_1 + w_1, \ldots, v_n + w_n)$
- **scalar product** or **dot product**:
  $\boldsymbol{v} \cdot \boldsymbol{w} = (v_1, \ldots, v_n) \cdot (w_1, \ldots, w_n) = v_1 \cdot w_1 + \ldots + v_n \cdot w_n$
- **vector length** or **norm**:
  $|\boldsymbol{v}| = |(v_1, \ldots, v_n)| = \sqrt{v_1 \cdot v_1 + \ldots + v_n \cdot v_n} = \sqrt{\boldsymbol{v} \cdot \boldsymbol{v}}$

examples: basic linear algebra

- $0.5 \cdot (1, 0, 0, 1) = (0.5, 0, 0, 0.5)$
- $(1, 0, 0, 1) + (0, 0, 1, 1) = (1, 0, 1, 2)$
- $(1, 0, 0, 1) \cdot (0, 0, 1, 1) = 1 \cdot 0 + 0 \cdot 0 + 0 \cdot 1 + 1 \cdot 1 = 1$
- $|(1, 0, 0, 1)| = \sqrt{1 \cdot 1 + 0 \cdot 0 + 0 \cdot 0 + 1 \cdot 1} = \sqrt{2}$

beware the ambiguous notation!

- multiplying two numbers: $a \cdot b$ or ab
- scaling a vector $a \cdot \boldsymbol{v}$ or $a\boldsymbol{v}$
- dot product between two vectors: $\boldsymbol{v} \cdot \boldsymbol{w}$ or $\boldsymbol{v}\boldsymbol{w}$
- sometimes, numbers and vectors are distinguished by setting the vectors in boldface ($\boldsymbol{x}$, $\boldsymbol{v}$) or by arrow notation ($\vec{x}$, $\vec{v}$)

simple linear algebra implementation

- naively, we could implement the basic vector operations in Python:
  - ```
    def scale(a, v):
        return [a*vk for vk in v]
    ```
  - ```
    def vsum(v, w):
        return [vk+wk for (vk,wk) in zip(v, w)]
    ```
  - ```
    def dot(v, w):
        return sum([vk*wk for (vk,wk) in zip(v, w)])
    ```
  - ```
    def vlength(v):
        return math.sqrt(dot(v, v))
    ```
- however, this is inefficient if the dimension of the vector space is high

linear algebra implementation: better

- **NumPy** and **SciPy** are Python libraries containing many mathematical functions
  - they are interlinked and typically installed together
  - scikit-learn relies on both of them
- they use specialized math libraries to make computations faster
  - e.g. BLAS for your processor or graphics card
- example with a 100 million dimension random vector:
  - my simple function `dot(v, v)` takes 81 seconds
  - `numpy.dot(v, v)` takes 0.15 seconds

NumPy linear algebra examples

```
>>> import numpy
>>> v1 = numpy.array([1, 0, 0, 1, 0])
>>> v2 = numpy.array([0, 2, 1, -2, 1])
>>> v1
array([1, 0, 0, 1, 0])
>>> v2
array([ 0,  2,  1, -2,  1])
>>> v1 + v2
array([ 1,  2,  1, -1,  1])
>>> 100 * v1
array([100,   0,   0, 100,   0])
>>> numpy.dot(v1, v2)
-2
>>> v1.dot(v2)
-2
>>> numpy.linalg.norm(v1)
1.4142135623730951
```

sparse vectors

- in NLP, feature vectors are a bit peculiar compared to some other fields (e.g. speech and image processing):
  - the vector spaces often have a very high dimension
  - in each feature vector, most of the entries are zero
  - ["prices", "fall"] $\rightarrow$ (0, 1, 0, . . . , 0, 1, 0, . . . , 0, 0, 0)
- **sparse vector**: keep track of non-zero entries only: `[(2, 1), (10, 1)]`
- in some cases, this saves memory and is much faster

sparse vectors in Python

- SciPy includes five different types of sparse vectors
- in scikit-learn, `DictVectorizer` and `CountVectorizer` create vectors of the class `csr_matrix`
- more on this when we discuss classifier implementation
- see also `http://docs.scipy.org/doc/scipy/reference/sparse.html`

matrices

- a **matrix** is a 2-dimensional array of numbers: a “list of lists”

$$\begin{bmatrix} 1 & 2 & 0 \\ -2 & 1 & 0 \end{bmatrix}$$

- note that a vector can be seen as a special case of a matrix: a **row** or a **column**

$$\begin{bmatrix} -2 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} -2 \\ 1 \\ 0 \end{bmatrix}$$

reasons for using matrices

- matrices have a geometric interpretation, as we'll see in a moment
- however, in this context we mainly care about them to speed up our programs
  - we can see matrices as collections of vectors
  - in Python, it's more efficient to carry out a small number of operations on large matrices than on many small vectors

basic matrix operations

the basic **elementwise** operations on matrices, similar to what we did for the vectors:

- **scaling**: multiply all the cells by some number

$$10 \cdot \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix}$$

- **addition** / **subtraction**:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} 11 & 22 \\ 33 & 44 \end{bmatrix}$$

matrix multiplication

- matrix **multiplication** is an extension of the dot product for vectors
- each cell in the new matrix is computed as the dot product between a row and a column:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} & & & \\ & & & \end{bmatrix}$$

matrix multiplication

- matrix **multiplication** is an extension of the dot product for vectors
- each cell in the new matrix is computed as the dot product between a row and a column:

$$\begin{bmatrix} \mathbf{1} & \mathbf{2} \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{10} & 20 \\ \mathbf{30} & 40 \end{bmatrix} = \begin{bmatrix} \mathbf{70} & \\ & \end{bmatrix}$$

matrix multiplication

- matrix **multiplication** is an extension of the dot product for vectors
- each cell in the new matrix is computed as the dot product between a row and a column:

$$\begin{bmatrix} \mathbf{1} & \mathbf{2} \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 10 & \mathbf{20} \\ 30 & \mathbf{40} \end{bmatrix} = \begin{bmatrix} 70 & \mathbf{100} \\ & \end{bmatrix}$$

matrix multiplication

- matrix **multiplication** is an extension of the dot product for vectors
- each cell in the new matrix is computed as the dot product between a row and a column:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} 10 & 20 \\ 30 & 40 \end{bmatrix} = \begin{bmatrix} 70 & 100 \\ 150 & 220 \end{bmatrix}$$

geometric interpretation of matrix multiplication

- as mentioned, we use matrix multiplication (and other matrix operations) mainly for efficiency in this course
  - a matrix multiplication instead of many dot products
- however, in geometry we can use matrix multiplication can be used to express many useful transformations
  - scaling
  - rotation
  - projection from 3D to 2D
  - ...

matrix multiplication in NumPy

```
A = numpy.array([[1, 2], [3, 4]])

B = numpy.array([[10, 20], [30, 40]])

print(A.dot(B))
```

overview

the perceptron revisited

basic linear algebra and its implementation in Python

converting features to numerical vectors

linear classifiers

the first step: mapping features to numerical vectors

- scikit-learn's learning methods works with features as **numbers**, not strings
- they can't directly use the feature dicts we have stored in `X`
- converting from string to numbers is the purpose of these lines:

```
vec = DictVectorizer()
Xe = vec.fit_transform(X)
```

X		Xe
{ 'label':'NP', ... }		1 0 0 0 ...
{ 'label':'PP', ... }		0 1 0 0 ...
{ 'label':'S', ... }	$\longrightarrow$	0 0 1 0 ...
...		...
...		...
...		...
{ 'label':'PP', ... }		0 1 0 0 ...

types of vectorizers

- a `DictVectorizer` converts from attribute–value dicts:

$$\begin{array}{ccc} \mathtt{X} & & \mathtt{Xe} \\ \begin{bmatrix} \texttt{\{ 'label':'NP', ... \}} \\ \texttt{\{ 'label':'PP', ... \}} \\ \texttt{\{ 'label':'S', ... \}} \\ \texttt{...} \\ \texttt{...} \\ \texttt{...} \\ \texttt{\{ 'label':'PP', ... \}} \end{bmatrix} & \longrightarrow & \begin{bmatrix} \texttt{1 0 0 0 ...} \\ \texttt{0 1 0 0 ...} \\ \texttt{0 0 1 0 ...} \\ \texttt{...} \\ \texttt{...} \\ \texttt{...} \\ \texttt{0 1 0 0 ...} \end{bmatrix} \end{array}$$

- a `CountVectorizer` converts from texts (after applying a tokenizer) or lists:

$$\begin{array}{ccc} \mathtt{X} & & \mathtt{Xe} \\ \begin{bmatrix} \texttt{"this is a text"} \\ \texttt{"here is another text"} \\ \texttt{...} \\ \texttt{...} \\ \texttt{...} \\ \texttt{...} \\ \texttt{"a cat on a mat"} \end{bmatrix} & \longrightarrow & \begin{bmatrix} \texttt{1 1 1 1 0 0 ...} \\ \texttt{0 1 0 1 1 1 ...} \\ \texttt{...} \\ \texttt{...} \\ \texttt{...} \\ \texttt{...} \\ \texttt{0 0 2 0 0 0 ...} \end{bmatrix} \end{array}$$

- a `TfidfVectorizer` is like a `CountVectorizer`, but also uses TF*IDF

what goes on in a DictVectorizer?

- each feature corresponds to one or more columns in the output matrix
- easy case: boolean and numerical features:

$$\overset{\textstyle X}{\begin{bmatrix} \texttt{\{ 'f1':False, 'f2':7 \}} \\ \texttt{\{ 'f1':True, 'f2':2 \}} \\ \texttt{\{ 'f1':False, 'f2':9 \}} \end{bmatrix}} \longrightarrow \begin{array}{c} Xe \\ \begin{bmatrix} 0 & 7 \\ 1 & 2 \\ 0 & 9 \end{bmatrix} \\ \begin{array}{cc} \texttt{f1} & \texttt{f2} \end{array} \end{array}$$

what goes on in a DictVectorizer?

- each feature corresponds to one or more columns in the output matrix
- easy case: boolean and numerical features:

$$\begin{array}{c} X \\ \begin{bmatrix} \texttt{\{ 'f1':False, 'f2':7 \}} \\ \texttt{\{ 'f1':True, 'f2':2 \}} \\ \texttt{\{ 'f1':False, 'f2':9 \}} \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} Xe \\ \begin{bmatrix} 0 & 7 \\ 1 & 2 \\ 0 & 9 \end{bmatrix} \\ \begin{array}{cc} \texttt{f1} & \texttt{f2} \end{array} \end{array}$$

- for string features, we reserve one column for each possible value: **one-hot encoding**
  - that is, we convert to booleans

$$\begin{array}{c} X \\ \begin{bmatrix} \texttt{\{ 'f1':'NP', 'f2':'in' \}} \\ \texttt{\{ 'f1':'NP', 'f2':'on' \}} \\ \texttt{\{ 'f1':'VP', 'f2':'in' \}} \end{bmatrix} \end{array} \longrightarrow \begin{array}{c} Xe \\ \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \\ \begin{array}{cccc} \texttt{f1:NP} & \texttt{f1:VP} & \texttt{f2:in} & \texttt{f2:on} \end{array} \end{array}$$

code example (DictVectorizer)

```
from sklearn.feature_extraction import DictVectorizer
X = [{'f1':'NP', 'f2':'in', 'f3':False, 'f4':7},
     {'f1':'NP', 'f2':'on', 'f3':True,  'f4':2},
     {'f1':'VP', 'f2':'in', 'f3':False, 'f4':9}]
vec = DictVectorizer()
Xe = vec.fit_transform(X)
print(Xe.toarray())

print(vec.vocabulary_)
```

code example (DictVectorizer)

```
from sklearn.feature_extraction import DictVectorizer
X = [{'f1':'NP', 'f2':'in', 'f3':False, 'f4':7},
     {'f1':'NP', 'f2':'on', 'f3':True,  'f4':2},
     {'f1':'VP', 'f2':'in', 'f3':False, 'f4':9}]
vec = DictVectorizer()
Xe = vec.fit_transform(X)
print(Xe.toarray())

print(vec.vocabulary_)
```

the result:

```
 [[ 1.  0.  1.  0.  0.  7.]
  [ 1.  0.  0.  1.  1.  2.]
  [ 0.  1.  1.  0.  0.  9.]]

 {'f4': 5, 'f2=in': 2, 'f1=NP': 0, 'f1=VP': 1, 'f2=on': 3, 'f3': 4}
```

CountVectorizers for document representation

- a CountVectorizer converts from documents
  - the document is a string or a list of tokens
- just like string features in a DictVectorizer, we use one-hot encoding so that each word type will correspond to one column

$$\overset{X}{\begin{bmatrix} \texttt{"example text"} \\ \texttt{"another text"} \end{bmatrix}} \longrightarrow \begin{array}{c} Xe \\ \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{bmatrix} \\ \begin{array}{ccc} \texttt{another} & \texttt{example} & \texttt{text} \end{array} \end{array}$$

code example (CountVectorizer)

```
X = ['example text',
     'another text']

vec = CountVectorizer()
Xe = vec.fit_transform(X)
print(Xe.toarray())

print(vec.vocabulary_)
```

code example (CountVectorizer)

```
X = ['example text',
     'another text']

vec = CountVectorizer()
Xe = vec.fit_transform(X)
print(Xe.toarray())

print(vec.vocabulary_)
```

the result:

```
[[0 1 1]
 [1 0 1]]

{'text': 2, 'example': 1, 'another': 0}
```

the vectorizer methods

- `fit`: look at the data, create the mapping
- `transform`: convert the data to numbers
- `fit_transform` = `fit` + `transform`

overview

the perceptron revisited

basic linear algebra and its implementation in Python

converting features to numerical vectors

linear classifiers

linear classifiers

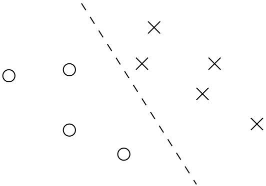
- a **linear classifier** is a classifier that is defined in terms of a scoring function like this

$$\text{score} = \boldsymbol{w} \cdot \boldsymbol{x}$$

- explanation of the parts:
  - $\boldsymbol{x}$ is a vector with features of what we want to classify (e.g. made with a `DictVectorizer`)
  - $\boldsymbol{w}$ is a vector representing which features the classifier thinks are important – this is just like our weight table before
  - $\cdot$ is the dot product between the two vectors
- there are two classes: **binary** classification
  - return the first class if the score > 0
  - ...otherwise the second class
- the essential idea: **features are scored independently**

geometric view

- geometrically, a linear classifier can be interpreted as separating the vector space into two regions with a line (plane, hyperplane)

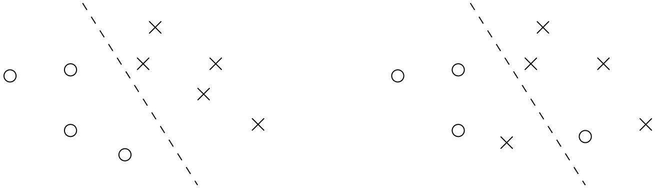


training linear classifiers

- the family of learning algorithms that create linear classifiers is quite large
  - perceptron, Naive Bayes, support vector machine, logistic regression/MaxEnt, ...
- their underlying theoretical motivations can differ a lot but in the end they all return a weight vector $\boldsymbol{w}$

linear separability

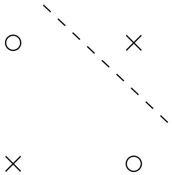
- a dataset is **linearly separable** if there exists a $\boldsymbol{w}$ that gives us perfect classification



- theorem: if the dataset is linearly separable, then the perceptron learning algorithm will find a separating $\boldsymbol{w}$ in a finite number of steps

a simple example of linear inseparability

very good	Positive
very bad	Negative
not good	Negative
not bad	Positive

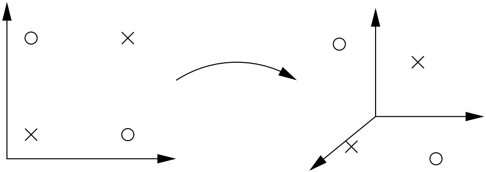


mapping into a larger vector space

- we may add **combinations** of features to make the dataset separable:

very good very-good	Positive
very bad very-bad	Negative
not good not-good	Negative
not bad not-bad	Positive

- from a geometrical viewpoint: we are creating a feature space with a higher dimensionality:



- lots of features $\rightarrow$ LOTS of combinations

coding a linear classifier using NumPy

```
class LinearClassifier(object):
    def predict(self, x):
        score = x.dot(self.w)
        if score >= 0.0:
            return self.positive_class
        else:
            return self.negative_class
```

better: handle all instances at the same time

```
class LinearClassifier(object):
    def predict(self, X):
        scores = X.dot(self.w)
        out = numpy.select([scores>=0.0, scores<0.0],
                           [self.positive_class,
                            self.negative_class])
        return out
```

an illustration of the steps

```
>>> import numpy

>>> scores = numpy.array([-1, 2, 3, -4, 5])

>>> scores >= 0
array([False,  True,  True, False,  True], dtype=bool)

>>> scores < 0
array([ True, False, False,  True, False], dtype=bool)

>>> numpy.select([scores >= 0, scores < 0], ["positive", "negative"])
array(['negative', 'positive', 'positive', 'negative', 'positive'],
      dtype='|S8')
```

perceptron reimplementation in NumPy

```
class NewPerceptron(LinearClassifier):

    def __init__(self, n_iter=10):
        self.n_iter = n_iter

    def fit(self, X, Y):
        # ... some initialization

        X = X.toarray() # convert sparse to dense
        n_features = X.shape[1]
        self.w = numpy.zeros( n_features )

        for i in range(self.n_iter):
            for x, y in zip(X, Y):

                score = self.w.dot(x)

                if score <= 0 and y == self.positive_class:
                    self.w += x
                elif score >= 0 and y == self.negative_class:
                    self.w -= x
```

a reformulation of the perceptron algorithm

- in many machine learning papers, the positive and negative class are implicitly represented as +1 and -1, respectively
- then the perceptron algorithm can be written a bit more compactly

```
class NewPerceptron2(LinearClassifier):
    # ...
    def fit(self, X, Y):
        # ... some initialization
        for i in range(self.n_iter):
            for x, y in zip(X, Y):
                score = self.w.dot(x)
                if y*score <= 0:
                    self.w += y*x
```

still too slow...

- this implementation uses NumPy's dense vectors
- with a large training set with lots of features, it may be better to use SciPy's sparse vectors
- however, $\boldsymbol{w}$ is a dense vector and I found it a bit tricky to mix sparse and dense vectors
- this is the best solution I've been able to come up with for the two operations $\boldsymbol{w} \cdot \boldsymbol{x}$ and $\boldsymbol{w} += \boldsymbol{x}$

```
def sparse_dense_dot(x, w):
    return numpy.dot(w[x.indices], x.data)

def add_sparse_to_dense(x, w, xw):
    w[x.indices] += xw*x.data
```

reimplementation with sparse vectors

```
class SparsePerceptron(LinearClassifier):

    # ...

    def fit(self, X, Y):
        # ... some initialization

        for i in range(self.n_iter):
            for x, y in zip(X, Y):

                score = sparse_dense_dot(x, self.w)

                if y*score <= 0:
                    add_sparse_to_dense(x, self.w, y)
```

comparison

- on my computer, with the data set we'll use in assignment 2:
  - dense vectors: 17 seconds
  - sparse vectors: 3 seconds

next lecture

- **optimization**: how to find the maximum or minimum of a mathematical function
- we will use this to introduce two other algorithms for training linear classifiers:
  - **support vector classifier** (`LinearSVC`)
  - **logistic regression** (`LogisticRegression`)
- overview of the second assignment